B-PERFect

Applying the PERF Approach to B Based System Developments

Alexandra Halchin[1,2(✉)], Abderrahmane Feliachi[1], Neeraj Kumar Singh[2], Yamine Ait-Ameur[2], and Julien Ordioni[1]

[1] RATP, ING/STF/QS, 54 Rue Roger Salengro, 94724 Fontenay-sous-Bois, France
{alexandra.halchin,abderrahmane.feliachi,julien.ordioni}@ratp.fr
[2] INPT-ENSEEIHT/IRIT, 2 Rue Charles Camichel, 31071 Toulouse, France
{Alexandra.Halchin,nsingh,yamine}@enseeiht.fr

Abstract. An independent safety assessment of railway software systems is performed by RATP (Régie Autonome des Transports Parisiens) for all safety-critical systems before their deployment in its network. Whenever possible, this activity is performed using the PERF approach (Proof Executed over a Retro-engineered Formal model). PERF is a methodology which handles formal verification of already developed software. This approach is applied to a variety of software systems, developed using languages such as SCADE, Ada or C. It provides an alternative verification that can be applied for the independent safety assessment of critical systems used by RATP. In this paper, we propose the B-PERFect method to generalize the application of the PERF approach for critical systems which are based on the B method. In particular, this paper focuses on transformation strategy from B language to the pivot language of PERF: HLL. HLL is a synchronous data-flow language equipped with formal verification techniques. The differences between B and HLL are pointed out and the translation process is presented in this regard.

Keywords: PERF · B method · HLL · Safety assessment · Translation

1 Introduction

For several years, RATP has been involved in the application of formal verification techniques to assess the safety level of railway systems. RATP pays a lot of attention to the safety of its deployed systems. This safety regime is implemented through a mandatory internal independent safety assessment of all safety-critical railway systems. It gave birth to a formal verification methodology called PERF [3]. It is an independent assessment that helps to double-check the safety of the developed software in addition to the verification performed by the software supplier.

PERF was designed to be applicable to any software system independently of their development processes and languages. By taking the source code of the developed software as the target of the verification, it ensures a complete

https://doi.org/10.1007/978-3-319-68499-4_11

language-agnostic and non-interference with the software supplier which drastically reduces any possible bias. It also allows for applying formal verification techniques to the safety assessment activity, which is not always achieved by the software supplier in its safety verification.

In order to support the different solutions of all RATP's suppliers, a number of translators were developed and integrated into PERF. These translators give a formal representation of the targeted source code in the PERF's pivot language HLL, a synchronous data-flow language, similar to Lustre, allowing to express, in the same formalism, the system behavior as well as safety requirements. The role of the translators is to give a semantics-preserving formalization of the software to be analyzed in HLL. Currently, translators for SCADE, Ada and C languages are integrated in the PERF tool chain.

In a similar vein, the B-PERFect project was initiated in order to investigate the applicability of PERF on software systems developed using the B method [1]. Software systems developed using B are valid by construction with respect to safety requirements. The idea behind the B-PERFect project is not to replace the formal verification process of B but to propose a verification alternative to be used for the internal independent safety assessment. This will not question the proof process of B. However, it may eventually reveal any error in the initial formalization of safety requirements.

This paper describes a new approach for software safety verification and gives an overview of a translation process from B0 (a subset of B language close to imperative programs) to HLL, the pivot language of PERF. Moreover, it shows the general architecture of the code generation process, including technical challenges related to tool development. Section 2 introduces the context of this work and motivates the proposed approach. The required background related to the B method and HLL is described in Sect. 3. Section 4 presents the general translation strategy. In Sect. 5, we present some related works. Finally, Sect. 6 concludes our work along with future directions.

2 Context

RATP operates one of the most complex urban multi-modal public transportation networks in the world. In the Parisian region, its network includes 16 metro lines, 2 RER (intercity trains) lines, 7 tramway lines and more than 300 bus lines; transporting not less than 10 Million passengers each day. RATP has built, throughout the years, a rich expertise not only in operating transportation networks, but also in the engineering of railway transportation systems. This expertise made RATP one of the world pioneers in metro automation and one of the experts in automating existing lines.

The growing demand of transportation capacity coupled with continuous advances in computer technology accelerates the obsolescence of existing systems. These factors, added to the improvement and modernization desires, have led RATP to upgrade its network by adopting integrated and upgradeable solutions, through partially or fully automated transportation systems. The coexistence of these different systems brings additional difficulties, particularly related

to the safety assessment of the railway systems which depend on the automation level of these systems. One major concern of RATP is to ensure the safety of any deployed system on the network during all the project phases.

In order to guarantee a better and more extensive safety analysis, RATP's engineering department (ING) relies, whenever possible, on rigorous verification methodologies based on formal methods. One of the first application of formal methods in an RATP project goes back to the late eighties were the Z method revealed a number of safety critical bugs for the SACEM system (RER A) which already passed the tests campaign. This successful application of formal methods led RATP to require their use by all safety-critical software systems suppliers. As a consequence, the development of the first driverless metro line in Paris (Line 14) in 1998 was realized using the B formal method. The safety of the system was proven by the construction which helped to remove all testing phases while guaranteeing a complete coverage.

The use of formal methods cannot be required by RATP anymore because, according to the regulations, this would promote some suppliers over the others. However, the use of a formal development method is still highly recommended by RATP to all its suppliers. In addition, an independent safety assessment is performed internally by RATP. RATP's opinion is that using formal methods independently of the supplier reveals usually more bugs than the simple verification of the supplier's testing campaign. Since the 2000s, RATP is working with different suppliers, using different development methods and languages. This heterogeneity requires RATP to master all its supplier methods and languages, which introduced a skill management difficulty with regards to the assessment process. The solution was to use a unified verification approach, pointed as an "ex post facto" proof, for the different projects which allows for the application of formal verification independently of the supplier's development language or method.

This situation was the starting point of the PERF (Proof Executed over a Retro engineered Formal model) methodology and its supporting team. The technique has been successfully used on Thales, Ansaldo and Alstom (ex-Areva TA) products, in charge of the Computer Based Interlocking Lines 1, 4, 8 & 12, the wayside and the on-board equipment of CBTC (Communication Based Train Control) Line 3, 5, 9 & 13 projects. PERF is now applied in every project, whenever possible, meaning essentially when the source language of the software is supported. This is currently the case for projects developed using C, Ada or Scade languages. The general workflow of the PERF methodology is given in Fig. 1. The real strength of PERF is its supporting tool chain, composed of translators, counter-example analyzers and SAT-based proof engines [17].

A number of projects keep using the B method for the development of safety-critical systems. In this case, the independent assessment is a bit more complicated and might be intrusive in some situations. Even though the formal verification performed by the B proof engines can be trusted, the validation of the safety properties can only be performed by cross-reading which, besides being a tedious task, may not be very effective. The idea of the B-PERFect project is to provide an independent alternative for the verification of the safety

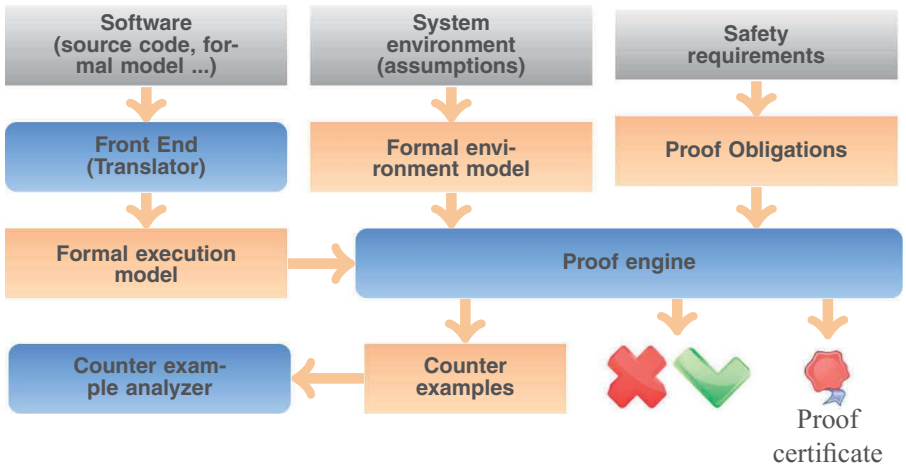

Fig. 1. The PERF verification workflow

properties on systems developed using the B method. The B code is transformed in a HLL formal execution model. To this model, the safety requirements targeted by the verification are added and the entire model is passed to the prover. By doing so, one can prove initial system properties which are expressed in natural language. The idea behind this is not to prove again the existent B code but to check if safety properties were modeled correctly in the initial code. The PERF approach makes this verification non intrusive and also supports the verification of the code generation process if needed. It will also help, in the context of heterogeneous systems, to apply a unified verification to all system components.

3 Background

B Method. The B method is a formal method based on first-order logic and set theory. It can handle a complete critical-software development process from specification to code [1]. The B development process is layered. Each layer corresponds to an abstraction level and the refinement provides the relation between layers. This method has proven its feasibility for large-scale industrial applications, particularly in railway domain [2].

Models are represented in B as machines. A machine contains state variables, instances of other machines, a state invariant, an initialization clause and operations acting on the defined variables. Generally, B project models represent a state transition system in which the initialization clause sets the initial values of variables and the operation clause specifies how variables are modified from one state to another. The invariant describes the safety properties of the model and is specified using predicate logic. The highest level of abstraction is the specification, a representation of functional requirements and the lowest one corresponds to the implementation where only programming-like constructs are allowed [7]. The refinement is the process of transformation from an abstract model into a concrete model specified in a subset of the B language: the B0 language, which can

be automatically translated into executable code [5,14,18,19]. Last level of refinement called implementation must be deterministic. For instance, parallel substitutions are not allowed, the type of variables must be scalar and modules are written in a procedural style. The advantage of using the B method is that it supports a correct by construction development approach which implies that each step of the development process can be proved if the target is a zero bug development.

B Development Example. As an example, the below implementation describes a simplified B machine which reads the input values from an external machine and computes the minimum of two variables. This example contains two B machines: `Utils_i` defines auxiliary operations and `Main_i` defines the main program. The `Main_i` machine represents an entry point of the execution. `Main` is an operation to select an order of the execution using defined operations in the imported machine. In the example, firstly, the operation `computeSum` is called that changes the state of the machine `Utils_i` as a side effect. The variable `xx` is initialized using the output of the operation `readVar`. This operation returns the value of a variable which is modified when `computeSum` is called. Finally, the minimum of two variables is computed using the operation `minimum`.

```
IMPLEMENTATION Main_i REFINES Main IMPORTS Utils
CONCRETE_VARIABLES xx,yy,rr
INVARIANT xx ∈ NAT ∧ yy ∈ NAT ∧ rr ∈ NAT
INITIALISATION xx:= 0 ; yy:= 0 ; rr:= 0
OPERATIONS
  Main =
   computeSum; xx <-- readVar; rr <-- minimum (xx , yy)
  END
END
```

Listing 1. Main Implementation of B Machine

```
IMPLEMENTATION Utils_i REFINES Utils
CONCRETE_VARIABLES sum
INVARIANT sum ∈ NAT
INITIALISATION sum:= 0
OPERATIONS
 rr <-- minimum (aa, bb) =
    IF aa >= bb THEN rr:= bb ELSE rr:= aa END;
 computeSum  =
  VAR ii IN ii:= 0;
     WHILE ii < 2 DO
         ii:= ii + 1; sum:= sum + ii;
     INVARIANT ii ∈ NAT ∧ ii ≤ 2
     VARIANT 2 - ii
     END
  END;
 rr <-- readVar =
     rr:= sum
END
```

Listing 2. Utils Implementation of B Machine

HLL, the Pivot Language of PERF. The PERF approach is built around HLL (High Level Language), a formal declarative and synchronous data flow language in the tradition of LUSTRE [11]. Models are defined by typed streams that can be composed using either temporal or data operators. Temporal operators can be used to describe clock-dependent expressions. The data operators, such as arithmetic, logical and array operators, are used to manipulate streams values. The declarative nature of the language makes it suitable for the definition of formal models as well as safety properties.

An HLL model is described by a number of sections containing type definitions, constant definitions, stream declarations and definitions, proof obligations, constraints and namespaces definitions. Streams can have integer or boolean values and they are interpreted in the mathematical sense, without any notion of side effects. The notion of sequentiality is absent, which means that the order of the items does not affect the meaning of the HLL model. A HLL project is organized in *namespaces* sections. Streams are declared in *declarations* blocks with type checking information, and their values are given in the *definitions* blocks. The *proof obligations* block contains a set of properties related to streams for verification purpose. *Constraints* expressions are used to reduce the domain definition of unbound input streams.

HLL Development Example. This section describes the HLL model that would result from translating the B example given above. The produced HLL model contains two namespaces, one corresponding to the translation of the `Main_i` machine and another for the translation of the imported machine `Utils_i`. For each B operation, a corresponding HLL namespace section is created, such as `"Main"` which contains the translation of the B operation Main.

```
1  Namespaces: "Main_i"{ // B: Main_i implementation
2  Declarations:
3  int "xx"; int "yy"; int "rr"; int "xx<0>"; int "yy<0>";int "rr
   <0>";
4  Definitions:  "xx<0>" := 0; "yy<0>" := 0; "rr<0>" := 0;
5  "xx" := "Main"::"xx<1>"; // B: xx <-- readVar;
6  "yy" := "Main"::"yy<0>";
7  "rr" := "Main"::"rr<1>"; // B: rr <-- minimum(xx,yy)
8  Namespaces: "Main"{ // B: Main operation
9  Declarations: int "xx<0>"; int "yy<0>"; int "rr<0>";
10 Definitions:
11 "xx<0>" := "Main_i"::"xx<0>"; // Maps the initial values of
   variables
12 "yy<0>" := "Main_i"::"yy<0>";
13 "rr<0>" := "Main_i"::"rr<0>";
14 "xx<1>" := "Utils_i<0>"::"readVar<0>"::"rr"; // Operation call
15 "rr<1>" := "Utils_i<0>"::"minimum<0>"::"rr"; // Operation call
16 }}
```

Listing 3. HLL Translation of Main Machine

HLL is an SSA language (Single State Assignment) since, in a model, a stream can be assigned only once. As stated in [8], when converting from a programming language to SSA form, assignments of a program variable are replaced with

assignments to new versions of the variable. Each B assignment will thus be translated to an HLL assignment with a new version of the modified variable. The value of the original variable is replaced by the value of the last known version of this variable.

```
17    Namespaces: "Utils_i<0>"{ // B:Utils_i implementation
18    Declarations: int "sum<0>";  int "sum<1>";
19    Definitions:  "sum<0>" := 0;
20    "sum<1>" := "computeSum<0>"::"sum";
21    Namespaces: "computeSum<0>"{ // First call of B: computeSum
      operation
22    Declarations:
23    int "sum<0>"; int "ii<0>"; int "ii<1>"; int "ii<2>"; int "sum";
24    Definitions:
25    "sum<0>" := "Utils_i<0>"::"sum<0>"; "ii<0>" := 0;
26    // While Loop - iter 0
27    "ii<1>" := "ii<0>" + 1;
28    "sum<1>":= "sum<0>" + "ii<1>";
29    "ii<2>" := if "ii<0>" < 2 then "ii<1>" else "ii<0>";
30    "sum<2>" := if "ii<0>" < 2 then "sum<1>" else "sum<0>";
31    //... Repeat the loop code with new index
32    "sum" := "sum<4>";
33    }
34    "readVar<0>"{ // First call of B: readVar operation
35    Declarations: int "rr";
36    Definitions: "rr":= "Utils_i<0>"::"sum<1>";
37    }
38    "minimum<0>"{ // First call of B: minimum operation
39    Declarations:
40    int "aa<0>";int "bb<0>";int "rr";int "rr<0>";int "rr<1>";int"rr
      <2>";
41    Definitions:
42    "aa<0>" := "Main_i":":Main"::"xx<1>"; //Mapping of input
      parameters
43    "bb<0>" := "Main_i"::"Main"::"yy<0>";
44    "rr<0>" := "bb<0>"; // IF block substitution
45    "rr<1>" := "aa<0>"; // ELSE block substitution
46    "rr<2>" := if "aa<0>" >= "bb<0>" then "rr<0>" else "rr<1>";//IF
      block
47    "rr" := "rr<2>";
48    }}
```

Listing 4. HLL Translation of Utils Machine

Line 3 defines the variables used for the translation of the machine `Main_i` with their corresponding type. Line 4 and lines 8–16 represent the computation done in blocks `INITIALISATION` and `OPERATIONS` of the B machine, respectively. In line 14, the output of the operation `readVar` is assigned to the local variable `"xx<1>"`. Note that state variables are necessary to memorize the final values of variables after the execution of the operation `Main` (lines 5–7). As the operation call `computeSum`, does not modify the state of variables in the machine `Main_i`, its translation is not present in the `Main` namespace. Lines 21–35 represent the translation of the first call of `computeSum`.

4 Translation Principles

Our work consists in translating concrete formal models based on B0 language in HLL. We propose a transformation strategy, allowing to obtain an equivalent HLL code which is further used for verification purposes. The goal of this work is to obtain HLL models which are behaviorally equivalent to B modules.

The semantic-preserving translation from B to HLL is not straightforward. The first issue to handle is the semantic mismatch between the two paradigms. Thus, a particular attention has to be given to several notions like variable values evolution and updates or loops behaviors. An example of such problems is illustrated in Listing 1. There, a B machine may have operations with side-effects, implicitly affecting the state of another B machine. For *Main_i* machine, the changes that occur to the variables in order to compute the value of the `sum` are transparent and not explicit. If the translation process does not follow the correct sequence of the variable changes, the generated HLL model may be erroneous. This kind of scenarios is very tricky to handle. It leads to incorrect HLL models and may hide problems related to safety. Figure 2 illustrates the general translation process made of three main steps: B parsing, preprocessing and code generation. In this paper, we focus on the code generation phase.

The first step of our approach generates an intermediate tree representation AST (Abstract Syntax Tree) of an input code by analyzing it syntactically and semantically. B0 is close to an imperative programming language and handles deterministic B instructions: concrete data (variables and constants), *SEES*, *USES* and *IMPORTS* clauses, and operation calls. Due to the semantics of the HLL language, the preprocessing step annotates the abstract syntax tree with additional information useful for variables evolution or loop transformation. This annotation defines an environment used and updated on the fly by each application of a translation rule. The last part of the process is the HLL code generation. Below, we give the relevant elements related to the code generation process we have set up. We have limited this description due to space limitation.

General Concepts. At the present time, we are interested in translating the IMPLEMENTATION module, the lowest level of a B project, in HLL. Since HLL proposes constructs to divide models in small units and to avoid naming conflicts, the initial B component structure can be preserved in the translation.

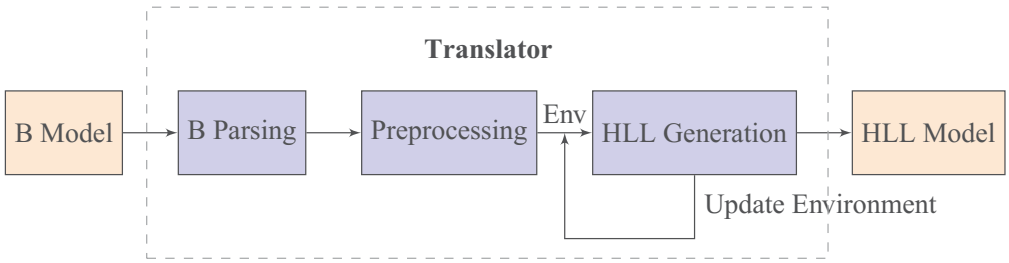

Fig. 2. Translation Workflow from B to HLL

Therefore, we propose to model B machines as HLL namespaces because both have a notion of variable scoping and structuring facilities which lead to a certain data encapsulation. Dependent machines obtained from IMPORTS, USES and SEES clauses must also be translated into HLL namespaces.

The language used in B expressions is essentially predicate logic and set theory. A B arithmetic expression is a mathematical formula that can contains constants, variables and operators. The supported arithmetic operators are: $+,-,\times,\div$. A predicate expression is evaluated to be true or false in B0 as branching conditions of *if substitutions* or in *while loops*. Except for division, the translation of B expressions and B predicates is straightforward because HLL provides the same quantifiers as B [15].

Sequence. In B0, a sequence represents an action which leads to the next action in a predetermined order. All B variables are translated in HLL variables with equivalent types. The link between B0 variables and HLL variables is very crucial for semantics preservation of the translation. This link is not very obvious. In B, variables may evolve during the execution of operations, whereas, in HLL, they correspond to data streams without memory and having a unique value during a cycle. However, our goal is to maintain memory state consistency between B and HLL representation. The HLL equivalent representation of a B variable `xx` is `"xx<i>"` for each occurrence `i` of this variable in left-hand side of an assignment. A new HLL variable is defined by the concatenation of the B variable name and its state evaluated in the translation context. While applying this renaming process, the following properties must be preserved: (i) all value changes of a variable shall be traced and (ii) generated code shall preserve the semantics of the B language. Therefore, the context in which a variable modification occurs is stored and associated to a variable.

Operations. In B language, the dynamic parts of the components are modeled by substitutions, which allow the modification of the data space of a model. Substitutions are used in INITIALISATION and OPERATIONS clauses of a B machine. The proposed transformation of B0 substitutions is based on the understanding of the semantic differences between HLL and B. The general form of an operation is: $out \leftarrow op\_name(in)$ where in and out can be variables or lists of variables representing the parameters of the operation op_name. Each B operation is translated in HLL as follows: inside the namespace associated to the translation of a machine we define a new namespace section which contains the translation of an operation. This namespace will have the same name as the original operation appended to an index, counting the different calls of the latter. Parameter passing is one of the crucial points for the semantics preservation when translating programs [4]. In the B language, parameters are passed by reference when calling operations. HLL does not support functions with non scalar types as it is used in common programming languages. In order to preserve the B semantics when transforming to HLL, the translation of B operation call is realized in two steps by separating the operation body substitutions translation

and the parameter mapping translation. Extra assignments are introduced in order to map the effective input parameters to formal input ones in an operation call namespace. This situation is illustrated in Listing 4, lines 42–43 where variables `"aa<0>","bb<0>"` have the role of formal input parameters of the namespace. The operation output it is transformed in a new assignment as shown in Listing 3, line 15.

If Conditions. Both languages provide *IF* construction with the difference that in HLL it is an expression where in B language it is a statement. In order to merge the information issued from different control flow branches, the translation is performed in two steps. First, the blocks of instructions of each branch are translated (Listing 4, lines 44–45), second, extra conditional HLL assignments are introduced taking into account the condition evaluated initially and the previous substitutions. In the example of `minimum` operation, this corresponds to line 46 of Listing 4.

While Loops. Unlike the B language, HLL does not support loop structures. Therefore, B loops should be flattened in the HLL model. The general form of a loop construct in B0 is `WHILE C DO S INVARIANT I VARIANT V END`, where `S` is a substitution, `C` is a boolean expression, `I` is a loop invariant and `V` is a variant that guarantees the loop termination. In B, `while loop` is a shorthand for writing the same block of instructions many times. A `while loop` must end after a finite number of iterations a variant is required. We propose to translate `while loop` as HLL `if expressions` repeated as many times as the maximum number of iterations needed to exit the loop. This information is extracted using the `VARIANT` clause. The substitution `S` is translated using HLL constructs. The translation of invariant is not explicitly required in the HLL code, but it could be modeled as HLL Proof Obligations or Constraints. In the example presented in Listing 2 the maximum number of iterations of the loop is 2, so the HLL translation process repeats according to it. In Listing 4, lines 26–30 show the translation of the first loop iteration. The fact that variables are expressed in function of condition and their previous value guarantees the correctness of the translation by value propagation even if the number of iterations is an over-approximation.

5 State of the Art

There are several works [4,18,19] focusing on code generation in many programming languages (i.e. C, Ada and Java) from B specifications. In [13], the authors present a set of translation rules from B to Java/SQL studied in the database domain. To increase the use of formal methods, a tool B2Jml [6] was developed to produce JML specifications from B models. Bonichon et al. [5]

have developed LLVM-based code generator that provides llvm executable code for B specification. Moreover, they have also developed a tool b2llvm to automate the code generation process. Furst et al. [10] proposed a code generator to produce C code from Event-B models. In Singh et al. [14], a tool supported code generator, namely EB2ALL, producing source code in many programming languages from verified Event-B specifications is described. Following similar principles, Ge et al. [15] have proposed an approach for translating Event-B models into HLL models. In fact, the main objective of this work is to produce C code from Event-B specification using an intermediate HLL representation. To our knowledge, the proposed translation approach from Event-B to HLL is not automated yet. Similarly, Petit-Doche et al. [16] reported an a posteriori approach for applying formal methods on the developed software, in which a translation strategy is proposed to transform SCADE code to HLL code. In [12], the authors present an approach based on the synchronous language SIGNAL [9] to validate system designs. SIGNAL formal models are generated from C/C++ programs using an SSA intermediate representation. Moreover, translators from C, ADA to HLL already exist. The used translation strategy is not a direct one. An intermediate imperative language is used as a pivot language. There, the goal is to avoid multiple translation steps and to master the whole translation process. It is important to observe that our approach is in similar vein in order to increase confidence in the generated code and promote the use of formal methods in industrial practices. In our work, we propose a translation strategy to produce HLL code from B specification covering the whole B project. Moreover, our approach also highlights the process of translation from a tool development point of view.

6 Conclusion

We study the applicability of PERF, an industrial toolset which allows the formal verification of systems independently of their development process, on software developed in B. This paper presents our approach to generate verifiable HLL code from an implementation described as B0 code. We focus on the core concepts to ensure semantics preservation when translating B0 implementations to HLL data-flow language. The semantic differences between the two studied languages are pointed out and a general translation scheme is proposed. We describe a translation process as well as a set of translation principles for the constructs that require a particular attention. Our initial ideas are already under development on a prototype tool for automatic translation. In this perspective, we have investigated the existing B parsers and BCompiler[1], an open source tool that offers complex parsing features for syntactical and semantical analysis.

Our future work consists in providing a formalization of the translation rules which shall cover the whole B components and constructs. The correctness of

[1] https://sourceforge.net/projects/bcomp/.

the translation is not studied in this paper. A possible starting point could be the definition of the semantics of both B and HLL in a unified framework and then check semantics preservation. Another possible extension of this work is to handle higher abstraction levels of the B developments in order to enrich the HLL model with lemmas or hints that might help the proof of properties.

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Behm, P., Benoit, P., Faivre, A., Meynadier, J.-M.: Météor: a successful application of B in a large project. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 369–387. Springer, Heidelberg (1999). doi:10.1007/3-540-48119-2_22
3. Benaissa, N., Bonvoisin, D., Feliachi, A., Ordioni, J.: The PERF approach for formal verification. In: Lecomte, T., Pinger, R., Romanovsky, A. (eds.) RSSRail 2016. LNCS, vol. 9707, pp. 203–214. Springer, Cham (2016). doi:10.1007/978-3-319-33951-1_15
4. Bert, D., Boulmé, S., Potet, M.-L., Requet, A., Voisin, L.: Adaptable translator of B specifications to embedded C programs. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 94–113. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45236-2_7
5. Bonichon, R., Déharbe, D., Lecomte, T., Medeiros, V.: LLVM-based code generation for B. In: Braga, C., Martí-Oliet, N. (eds.) SBMF 2014. LNCS, vol. 8941, pp. 1–16. Springer, Cham (2015). doi:10.1007/978-3-319-15075-8_1
6. Cataño, N., Wahls, T., Rueda, C., Rivera, V., Yu, D.: Translating B machines to JML specifications. In: SAC 2012, pp. 1271–1277. ACM (2012)
7. ClearSy: Atelier B user manual version 4.0 (2009)
8. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst. **13**(4), 451–490 (1991)
9. Espresso: Polychrony tool. http://www.irisa.fr/espresso/Polychrony
10. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code generation for Event-B. In: Albert, E., Sekerinski, E. (eds.) IFM 2014. LNCS, vol. 8739, pp. 323–338. Springer, Cham (2014). doi:10.1007/978-3-319-10181-1_20
11. Halbwachs, N., Caspi, P., Raymond, P., Pilaud, D.: The synchronous data flow programming language LUSTRE. Proc. IEEE **79**(9), 1305–1320 (1991)
12. Kalla, H., Talpin, J.P., Berner, D., Besnard, L.: Automated translation of C/C++ models into a synchronous formalism. In: ECBS 2006. pp. 9–436, March 2006
13. Mammar, A., Laleau, R.: From a B formal specification to an executable code: application to the relational database domain. Info. Soft. Technol. **48**(4), 253–279 (2006)
14. Méry, D., Singh, N.K.: Automatic code generation from EVENT-B models. In: SoICT 2011, pp. 179–188. ACM (2011)
15. Ge, N., Dieumegard, A., Jenn, E., Voisin, L.: Correct-by-construction specification to verified code. Ada-Europe 2017 (2017)
16. Petit-Doche, M., Breton, N., Courbis, R., Fonteneau, Y., Güdemann, M.: Formal verification of industrial critical software. In: Núñez, M., Güdemann, M. (eds.) FMICS 2015. LNCS, vol. 9128, pp. 1–11. Springer, Cham (2015). doi:10.1007/978-3-319-19458-5_1

17. Prasad, M.R., Biere, A., Gupta, A.: A survey of recent advances in SAT-based formal verification. Int. J. Softw. Tools Technol. Transf. **7**(2), 156–173 (2005)
18. Storey, A.C., Haughton, H.P.: A strategy for the production of verifiable code using the B Method. In: Naftalin, M., Denvir, T., Bertran, M. (eds.) FME 1994. LNCS, vol. 873, pp. 346–365. Springer, Heidelberg (1994). doi:10.1007/3-540-58555-9_104
19. Tatibouët, B., Requet, A., Voisinet, J.-C., Hammad, A.: Java card code generation from B specifications. In: Dong, J.S., Woodcock, J. (eds.) ICFEM 2003. LNCS, vol. 2885, pp. 306–318. Springer, Heidelberg (2003). doi:10.1007/978-3-540-39893-6_18